Advanced Topics in Numerical Analysis: High Performance Computing
MATH-GA 2012.003 and CSCI-GA 2945.003

Georg Stadler
Courant Institute, NYU
stadler@cims.nyu.edu

Spring 2015, Monday, 5:10–7:00PM, WWH #202

Mar. 2, 2015
Outline

Organization issues

Last week summary

Datatypes and non-blocking MPI calls

Git—repository systems

MPI Collectives (outlook)
Organization issues

- Homework is due next week–I will talk about how to hand it in using git today (and also do a writeup).
- Homework questions?
- I’ve some project suggestions.
Outline

Organization issues

Last week summary

Datatypes and non-blocking MPI calls

Git—repository systems

MPI Collectives (outlook)
Programming models

- Flynn’s taxonomy:
  - Single instruction–single data (SISD)
  - Single instruction–multiple data (SIMD)
  - Multiple instruction–multiple data (MIMD)

- Distributed memory vs. shared memory parallelism

- Programming models: OpenMP vs. Message passing interface (MPI); and combinations thereof
MPI parallelism

MIMD or SPMD (single program–multiple data) parallelism: Every processor core runs through same instruction set but has a different mpirank.

- Initialization and Finishing: MPI_Init(), MPI_Finalize();
- What’s the “context”, i.e., my environment? Find the number of processes and my mpirank: MPI_Comm_Size, MPI_Comm_Rank; MPI_COMM_WORLD: Default handle for all processes
- Cooperative (blocking) point-to-point communications:
  MPI_Send(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator),
  MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)
Outline

Organization issues

Last week summary

Datatypes and non-blocking MPI calls

Git—repository systems

MPI Collectives (outlook)
MPI datatypes

Need to specify MPI datatypes:

- MPI_INT, MPI_DOUBLE
- MPI_LONG
- MPI_UNSIGNED
- ...

To create user-specified data types:

- MPI_Type_continuous
- MPI_Type_vector
Blocking vs. non-blocking Send/Recv

- MPI_Send, MPI_Recv are blocking, i.e., they only return when they have finished sending/receiving data. In particular, the buffers can be used immediately again.

- Non-blocking communication does not block; the success of the communication must be checked:

  MPI_Isend(void *buffer, int count, MPI_Datatype datatype, int dest, int tag , MPI_Comm comm, MPI_Request *request)

  MPI_Irecv((void *buffer, int count, MPI_Datatype datatype, int dest, int tag , MPI_Comm comm, MPI_Request *request))

  MPI_Wait(MPI_Request *request, MPI_Status *status)
Outline

Organization issues

Last week summary

Datatypes and non-blocking MPI calls

Git—repository systems

MPI Collectives (outlook)
A Version Control System (VCS) is an integrated fool-proof framework for

- Backup and Restore
- Short and long-term undo
- Tracking changes
- Synchronization
- Collaborating
- Sandboxing

... with minimal overhead.
Local Version Control Systems

Conventional version control systems provide some of these features by making a local database with all changes made to files.

![Diagram of Local Computer with Checkout, Version Database, and file versions]

Any file can be recreated by getting changes from the database and patching them up.
Centralized Version Control Systems

To enable synchronization and collaborative features the database is stored on a central VCS server, where everyone works in the same database.

Introduces problems: single point of failure, inability to work offline.
Distributed Version Control Systems

To overcome problems related to centralization, distributed VCSs (DVCSs) were invented. Keeping a complete copy of database in every working directory.

Actually the most **simple** and most **powerful** implementation of any VCS.
The simplest use of Git:

- **Modify** files in your *working directory*.
- **Stage** the files, adding snapshots of them to your *staging area*.
- **Commit**, takes files in the staging area and stores that snapshot permanently to your *Git directory*.
The three basic states of files in your Git repository:
Git Basics - Commits

Each commit in the git directory holds a snapshot of the files that were staged and thus went into that commit, along with author information.

Each and every commit can always be looked at and retrieved.
Files in your working directory can be in four different states in relation to the current commit.
In Git all remotes are equal.

A *remote* in Git is nothing more than a link to another git directory.
The easiest commands to get started working with a remote are:

- **clone**: Cloning a remote will make a complete local copy.
- **pull**: Getting changes from a remote.
- **push**: Sending changes to a remote.

Fear not! We are starting to get into more advanced topics. So let's look at some examples.
Basic advantages of using Git:

- Nearly every operation is local.
- Committed snapshots are always kept.
- Strong support for non-linear development.
Hands-on - First-Time Git Setup

Before using Git for the first time:

Pick your identity

$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com

Check your settings

$ git config --list

Get help

$ git help <verb>
Hands-on - Getting started with a bare remote server

Using a Git server (ie. no working directory / *bare* repository) is the analogue to a regular centralized VCS in Git.
Hands-on - Getting started with remote server

When the remote server is set up with an initialized Git directory you can simply *clone* the repository:

**Cloning a remote repository**

```
$ git clone <repository>
```

You will then get a complete local copy of that repository, which you can edit.
With your local working copy you can make any changes to the files in your working directory as you like. When satisfied with your changes you add any modified or new files to the staging area using `add`:

Adding files to the staging area

```bash
$ git add <filepattern>
```
Finally to commit the files in the staging area you run `commit` supplying a `commit message`.

Committing staging area to the repository

```
$ git commit -m <msg>
```

Note that so far **everything is happening locally** in your working directory.
Hands-on - Getting started with remote server

To share your commits with the remote you invoke the push command:

**Pushing local commits to the remote**

$ git push

To receive changes that other people have pushed to the remote server you can use the pull command:

**Pulling remote commits to the local working directory**

$ git pull

And thats it.
Hands-on - Summary

Summary of a minimal Git workflow:

- clone remote repository
- add your changes to the staging area
- commit those changes to the git directory
- push your changes to the remote repository
- pull remote changes to your local working directory
More advanced topics

Git is a powerful and flexible DVCS. Some very useful, but a bit more advanced features include:

▶ Branching
▶ Merging
▶ Tagging
▶ Rebasing
Some good Git sources for information:

- Pro Git - [http://progit.org/](http://progit.org/)
- Git Reference - [http://gitref.org/](http://gitref.org/)
- GitHub - [http://github.com/](http://github.com/)
- Git from the bottom up - [http://ftp.newartisans.com/pub/git.from.bottom.up.pdf](http://ftp.newartisans.com/pub/git.from.bottom.up.pdf)
- Understanding Git Conceptually - [http://www.eecs.harvard.edu/~cduan/technical/git/](http://www.eecs.harvard.edu/~cduan/technical/git/)
- Git Immersion - [http://gitimmersion.com/](http://gitimmersion.com/)
Applications

GUls for Git:

- GitX (MacOS) - http://gitx.frim.nl/
- Giggle (Linux) - http://live.gnome.org/giggle
Outline

Organization issues

Last week summary

Datatypes and non-blocking MPI calls

Git—repository systems

MPI Collectives (outlook)
MIP Collectives

- MPI_Broadcast
- MPI_Gather
- MPI_Reduce
- MPI_Allreduce